Table of Contents

Introduction	1.1
Preface	1.2
Use the Tools Available	1.3
Style	1.4
Considering Safety	1.5
Considering Maintainability	1.6
Considering Portability	1.7
Considering Threadability	1.8
Considering Performance	1.9
Enable Scripting	1.10
Further Reading	1.11
Final Thoughts	1.12

cppbestpractices

gitter join chat

Collaborative Collection of C++ Best Practices

This document is available as a download via gitbook

For more information please see the Preface.

This book has inspired an O'Reilly video: Learning C++ Best Practices

Preface

C++ Best Practices: A Forkable Coding Standards Document

This document is meant to be a collaborative discussion of the best practices in C++. It complements books such as *Effective C++* (Meyers) and *C++ Coding Standards* (Alexandrescu, Sutter). We fill in some of the lower level details that they don't discuss and provide specific stylistic recommendations while also discussing how to ensure overall code quality.

In all cases brevity and succinctness is preferred. Examples are preferred for making the case for why one option is preferred over another. If necessary, words will be used.

C++ Best Practices by Jason Turner is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Disclaimer

This document is based on my personal experiences. You are not supposed to agree with it 100%. It exists as a book on GitHub so that you can fork it for your own uses or submit back proposed changes for everyone to share.

This book has inspired an O'Reilly video: Learning C++ Best Practices

Use The Tools Available

An automated framework for executing these tools should be established very early in the development process. It should not take more than 2-3 commands to checkout the source code, build, and execute the tests. Once the tests are done executing, you should have an almost complete picture of the state and quality of the code.

Source Control

Source control is an absolute necessity for any software development project. If you are not using one yet, start using one.

- GitHub - allows for unlimited public repositories, must pay for a private repository.
- Bitbucket - allows for unlimited private repositories with up to 5 collaborators, for free.
- SourceForge - open source hosting only.
- GitLab - allows for unlimited public and private repositories, unlimited CI Runners included, for free.
- Visual Studio Online (http://www.visualstudio.com/what-is-visual-studio-online-vs) - allows for unlimited public repositories, must pay for private repository. Repositories can be git or TFVC. Additionally: Issue tracking, project planning (multiple Agile templates, such as SCRUM), integrated hosted builds, integration of all this into Microsoft Visual Studio. Windows only.

Build Tool

Use an industry standard widely accepted build tool. This prevents you from reinventing the wheel whenever you discover / link to a new library / package your product / etc. Examples include:

- CMake
  - Consider: https://github.com/sakra/cotire/ for build performance
  - Consider: https://github.com/toeb/cmakepp for enhanced usability
  - Utilize: https://cmake.org/cmake/help/v3.6/command/target_compile_features.html for C++ standard flags
  - Consider: https://github.com/cheshirekow/cmake_format for automatic formatting of your CMakeLists.txt
- Conan - a crossplatform dependency manager for C++
- C++ Archive Network (CPPAN) - a crossplatform dependency manager for C++

- Waf
- FASTBuild
- Ninja - can greatly improve the incremental build time of your larger projects. Can be used as a target for CMake.
- Bazel - Note: MacOS and Linux only.
- gyp - Google's build tool for chromium.
- maiken - Crossplatform build tool with Maven-esque configuration style.
- Qt Build Suite - Crossplatform build tool From Qt.
- meson - Open source build system meant to be both extremely fast, and, even more importantly, as user friendly as possible.
- premake
- hunter - CMake driven cross-platform package manager for C/C++

Remember, it's not just a build tool, it's also a programming language. Try to maintain good clean build scripts and follow the recommended practices for the tool you are using.

Continuous Integration

Once you have picked your build tool, set up a continuous integration environment.

Continuous Integration (CI) tools automatically build the source code as changes are pushed to the repository. These can be hosted privately or with a CI host.

- Travis CI
  - works well with C++
  - designed for use with GitHub
  - free for public repositories on GitHub
- AppVeyor
  - supports Windows, MSVC and MinGW
  - free for public repositories on GitHub
- Hudson CI / Jenkins CI
  - Java Application Server is required
  - supports Windows, OS X, and Linux
  - extendable with a lot of plugins
- TeamCity
  - has a free option for open source projects
- Decent CI
  - simple ad-hoc continuous integration that posts results to GitHub
  - supports Windows, OS X, and Linux
  - used by ChaiScript
- Visual Studio Online (http://www.visualstudio.com/what-is-visual-studio-online-vs)

  - Tightly integrated with the source repositories from Visual Studio Online
  - Uses MSBuild (Visual Studio's build engine), which is available on Windows, OS X and Linux
  - Provides hosted build agents and also allows for user-provided build agents
  - Can be controlled and monitored from within Microsoft Visual Studio
  - On-Premise installation via Microsoft Team Foundation Server
- GitLab
  - use custom Docker images, so can be used for C++
  - has free shared runners
  - has trivial processing of result of coverage analyze

If you have an open source, publicly-hosted project on GitHub:

- go enable Travis Ci and AppVeyor integration right now. We'll wait for you to come back. For a simple example of how to enable it for your C++ CMake-based application, see here: https://github.com/ChaiScript/ChaiScript/blob/master/.travis.yml
- enable one of the coverage tools listed below (Codecov or Coveralls)
- enable Coverity Scan

These tools are all free and relatively easy to set up. Once they are set up you are getting continuous building, testing, analysis and reporting of your project. For free.

Compilers

Use every available and reasonable set of warning options. Some warning options only work with optimizations enabled, or work better the higher the chosen level of optimization is, for example `-Wnull-dereference` with GCC.

You should use as many compilers as you can for your platform(s). Each compiler implements the standard slightly differently and supporting multiple will help ensure the most portable, most reliable code.

GCC / Clang

`-Wall -Wextra -Wshadow -Wnon-virtual-dtor -pedantic`

- `-Wall -Wextra` reasonable and standard
- `-Wshadow` warn the user if a variable declaration shadows one from a parent context
- `-Wnon-virtual-dtor` warn the user if a class with virtual functions has a non-virtual destructor. This helps catch hard to track down memory errors
- `-Wold-style-cast` warn for c-style casts
- `-Wcast-align` warn for potential performance problem casts

- `-Wunused` warn on anything being unused
- `-Woverloaded-virtual` warn if you overload (not override) a virtual function
- `-Wpedantic` warn if non-standard C++ is used
- `-Wconversion` warn on type conversions that may lose data
- `-Wsign-conversion` warn on sign conversions
- `-Wmisleading-indentation` warn if identation implies blocks where blocks do not exist
- `-Wduplicated-cond` warn if `if` / `else` chain has duplicated conditions
- `-Wduplicated-branches` warn if `if` / `else` branches have duplicated code
- `-Wlogical-op` warn about logical operations being used where bitwise were probably wanted
- `-Wnull-dereference` warn if a null dereference is detected
- `-Wuseless-cast` warn if you perform a cast to the same type
- `-Wdouble-promotion` warn if `float` is implicit promoted to `double`
- `-Wformat=2` warn on security issues around functions that format output (ie `printf`)

Consider using `-Weverything` and disabling the few warnings you need to on Clang

`-Weffc++` warning mode can be too noisy, but if it works for your project, use it also.

MSVC

`/permissive-` - Enforces standards conformance.

`/W4 /W14640` - use these and consider the following (see descriptions below)

- `/W4` All reasonable warnings
- `/w14242` 'identfier': conversion from 'type1' to 'type1', possible loss of data
- `/w14254` 'operator': conversion from 'type1:field_bits' to 'type2:field_bits', possible loss of data
- `/w14263` 'function': member function does not override any base class virtual member function
- `/w14265` 'classname': class has virtual functions, but destructor is not virtual instances of this class may not be destructed correctly
- `/w14287` 'operator': unsigned/negative constant mismatch
- `/we4289` nonstandard extension used: 'variable': loop control variable declared in the for-loop is used outside the for-loop scope
- `/w14296` 'operator': expression is always 'boolean_value'
- `/w14311` 'variable': pointer truncation from 'type1' to 'type2'
- `/w14545` expression before comma evaluates to a function which is missing an argument list
- `/w14546` function call before comma missing argument list
- `/w14547` 'operator': operator before comma has no effect; expected operator with side-

effect

- `/w14549` 'operator': operator before comma has no effect; did you intend 'operator'?
- `/w14555` expression has no effect; expected expression with side-effect
- `/w14619` pragma warning: there is no warning number 'number'
- `/w14640` Enable warning on thread un-safe static member initialization
- `/w14826` Conversion from 'type1' to 'type_2' is sign-extended. This may cause unexpected runtime behavior.
- `/w14905` wide string literal cast to 'LPSTR'
- `/w14906` string literal cast to 'LPWSTR'
- `/w14928` illegal copy-initialization; more than one user-defined conversion has been implicitly applied

Not recommended

- `/Wall` - Also warns on files included from the standard library, so it's not very useful and creates too many extra warnings.

General

Start with very strict warning settings from the beginning. Trying to raise the warning level after the project is underway can be painful.

Consider using the *treat warnings as errors* setting. `/Wx` with MSVC, `-Werror` with GCC / Clang

LLVM-based tools

LLVM based tools work best with a build system (such as cmake) that can output a compile command database, for example:

```
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .
```

If you are not using a build system like that, you can consider Build EAR which will hook into your build system and generate a compile command database for you.

CMake now also comes with built-in support for calling `clang-tidy` during normal compilation.

- include-what-you-use, example results
- clang-modernize, example results
- clang-check
- clang-tidy

Static Analyzers

The best bet is the static analyzer that you can run as part of your automated build system. Cppcheck and clang meet that requirement for free options.

Coverity Scan

Coverity has a free (for open source) static analysis toolkit that can work on every commit in integration with Travis CI and AppVeyor.

PVS-Studio

PVS-Studio is a tool for bug detection in the source code of programs, written in C, C++ and C#. It is free for personal academic projects, open source non-commercial projects and independent projects of individual developers. It works in Windows and Linux environment.

Cppcheck

Cppcheck is free and open source. It strives for 0 false positives and does a good job at it. Therefore all warnings should be enabled: `--enable=all`

Notes:

- For correct work it requires well formed path for headers, so before usage don't forget to pass: `--check-config`.
- Finding unused headers does not work with `-j` more than 1.
- Remember to add `--force` for code with a lot number of `#ifdef` if you need check all of them.

CppDepend

CppDepend Simplifies managing a complex C/C++ code base by analyzing and visualizing code dependencies, by defining design rules, by doing impact analysis, and comparing different versions of the code. It's free for OSS contributors.

Clang's Static Analyzer

Clang's analyzer's default options are good for the respective platform. It can be used directly from CMake. They can also be called via clang-check and clang-tidy from the LLVM-based Tools.

Also, CodeChecker is available as a front-end to clang's static analysis.

`clang-tidy` can be easily used with Visual Studio via the Clang Power Tools extension.

MSVC's Static Analyzer

Can be enabled with the `/analyze` command line option. For now we will stick with the default options.

Flint / Flint++

Flint and Flint++ are linters that analyze C++ code against Facebook's coding standards.

OCLint

OCLint is a free, libre and open source static code analysis tool for improving quality of C++ code in many different ways.

ReSharper C++ / CLion

Both of these tools from JetBrains offer some level of static analysis and automated fixes for common things that can be done better. They have options available for free licenses for open source project leaders.

Cevelop

The Eclipse based Cevelop IDE has various static analysis and refactoring / code fix tools available. For example, you can replace macros with C++ `constexprs`, refactor namespaces (extract/inline `using`, qualify name), and refactor your code to C++11's uniform initialization syntax. Cevelop is free to use.

Qt Creator

Qt Creator can plug into the clang static analyzer.

clazy

clazy is a clang based tool for analyzing Qt usage.

Runtime Checkers

Code Coverage Analysis

A coverage analysis tool shall be run when tests are executed to make sure the entire application is being tested. Unfortunately, coverage analysis requires that compiler optimizations be disabled. This can result in significantly longer test execution times.

- Codecov
  - integrates with Travis CI and AppVeyor
  - free for open source projects
- Coveralls
  - integrates with Travis CI and AppVeyor
  - free for open source projects
- LCOV
  - very configurable
- Gcovr
- kcov
  - integrates with codecov and coveralls
  - performs code coverage reporting without needing special compiler flags, just by instrumenting debug symbols.

Valgrind

Valgrind is a runtime code analyzer that can detect memory leaks, race conditions, and other associated problems. It is supported on various Unix platforms.

Dr Memory

Similar to Valgrind. http://www.drmemory.org

GCC / Clang Sanitizers

These tools provide many of the same features as Valgrind, but built into the compiler. They are easy to use and provide a report of what went wrong.

- AddressSanitizer
- MemorySanitizer
- ThreadSanitizer
- UndefinedBehaviorSanitizer

Fuzzy Analyzers

If your project accepts user defined input, considering running a fuzzy input tester.

Both of these tools use coverage reporting to find new code execution paths and try to breed novel inputs for your code. They can find crashes, hangs, and inputs you didn't know were considered valid.

- american fuzzy lop
- LibFuzzer
- KLEE - Can be used to fuzz individual functions

Control Flow Guard

MSVC's Control Flow Guard adds high performance runtime security checks.

Checked STL Implementations

- `_GLIBCXX_DEBUG` with GCC's implementation libstdc++ implementation. See Krister's blog article.

Ignoring Warnings

If it is determined by team consensus that the compiler or analyzer is warning on something that is either incorrect or unavoidable, the team will disable the specific error to as localized part of the code as possible.

Be sure to reenable the warning after disabling it for a section of code. You do not want your disabled warnings to leak into other code.

Testing

CMake, mentioned above, has a built in framework for executing tests. Make sure whatever build system you use has a way to execute tests built in.

To further aid in executing tests, consider a library such as Google Test, Catch, CppUTest or Boost.Test to help you organize the tests.

Unit Tests

Unit tests are for small chunks of code, individual functions which can be tested standalone.

Integration Tests

There should be a test enabled for every feature or bug fix that is committed. See also Code Coverage Analysis. These are tests that are higher level than unit tests. They should still be limited in scope to individual features.

Negative Testing

Don't forget to make sure that your error handling is being tested and works properly as well. This will become obvious if you aim for 100% code coverage.

Debugging

uftrace

uftrace can be used to generating function call graphs of a program execution

rr

rr is a free (open source) reverse debugger that supports C++.

Other Tools

Lizard

Lizard provides a very simple interface for running complexity analysis against a C++ codebase.

Metrix++

Metrix++ can identify and report on the most complex sections of your code. Reducing complex code helps you and the compiler understand it better and optimize it better.

ABI Compliance Checker

ABI Compliance Checker (ACC) can analyze two library versions and generates a detailed compatibility report regarding API and C++ ABI changes. This can help a library developer spot unintentional breaking changes to ensure backward compatibility.

CNCC

Customizable Naming Convention Checker can report on identifiers in your code that do not follow certain naming conventions.

ClangFormat

ClangFormat can check and correct code formatting to match organizational conventions automatically. Multipart series on utilizing clang-format.

SourceMeter

SourceMeter offers a free version which provides many different metrics for your code and can also call into cppcheck.

Bloaty McBloatface

Bloaty McBloatface is a binary size analyzer/profiler for unix-like platforms

Style

Consistency is the most important aspect of style. The second most important aspect is following a style that the average C++ programmer is used to reading.

C++ allows for arbitrary-length identifier names, so there's no reason to be terse when naming things. Use descriptive names, and be consistent in the style.

- `CamelCase`
- `snake_case`

are common examples. *snake_case* has the advantage that it can also work with spell checkers, if desired.

Establishing A Style Guideline

Whatever style guidelines you establish, be sure to implement a `.clang-format` file that specifies the style you expect. While this cannot help with naming, it is particularly important for an open source project to maintain a consistent style.

Every IDE and many editors have support for clang-format built in or easily installable with an add-in.

- VSCode https://marketplace.visualstudio.com/items?itemName=xaver.clang-format
- VisualStudio https://marketplace.visualstudio.com/items?itemName=LLVMExtensions.ClangFormat#review-details
- Resharper++: https://www.jetbrains.com/help/resharper/2017.2/Using_Clang_Format.html
- Vim
  - https://github.com/rhysd/vim-clang-format
  - https://github.com/chiel92/vim-autoformat
- XCode: https://github.com/travisjeffery/ClangFormat-Xcode

Common C++ Naming Conventions

- Types start with upper case: `MyClass` .
- Functions and variables start with lower case: `myMethod` .
- Constants are all upper case: `const double PI=3.14159265358979323;` .

C++ Standard Library (and other well-known C++ libraries like Boost) use these guidelines:

- Macro names use upper case with underscores: `INT_MAX`.
- Template parameter names use camel case: `InputIterator`.
- All other names use snake case: `unordered_map`.

Distinguish Private Object Data

Name private data with a `m_` prefix to distinguish it from public data. `m_` stands for "member" data.

Distinguish Function Parameters

The most important thing is consistency within your codebase; this is one possibility to help with consistency.

Name function parameters with an `t_` prefix. `t_` can be thought of as "the", but the meaning is arbitrary. The point is to distinguish function parameters from other variables in scope while giving us a consistent naming strategy.

Any prefix or postfix can be chosen for your organization. This is just one example. *This suggestion is controversial, for a discussion about it see issue #11.*

```
struct Size
{
  int width;
  int height;

  Size(int t_width, int t_height) : width(t_width), height(t_height) {}
};

// This version might make sense for thread safety or something,
// but more to the point, sometimes we need to hide data, sometimes we don't.
class PrivateSize
{
  public:
    int width() const { return m_width; }
    int height() const { return m_height; }
    PrivateSize(int t_width, int t_height) : m_width(t_width), m_height(t_height) {}

  private:
    int m_width;
    int m_height;
};
```

Don't Name Anything Starting With `_`

If you do, you risk colliding with names reserved for compiler and standard library implementation use:

http://stackoverflow.com/questions/228783/what-are-the-rules-about-using-an-underscore-in-a-c-identifier

Well-Formed Example

```
class MyClass
{
public:
  MyClass(int t_data)
    : m_data(t_data)
  {
  }

  int getData() const
  {
    return m_data;
  }

private:
  int m_data;
};
```

Enable Out-of-Source-Directory Builds

Make sure generated files go into an output folder that is separate from the source folder.

Use `nullptr`

C++11 introduces `nullptr` which is a special value denoting a null pointer. This should be used instead of `0` or `NULL` to indicate a null pointer.

Comments

Comment blocks should use `//`, not `/* */`. Using `//` makes it much easier to comment out a block of code while debugging.

```
// this function does something
int myFunc()
{
}
```

To comment out this function block during debugging we might do:

```
/*
// this function does something
int myFunc()
{
}
*/
```

which would be impossible if the function comment header used `/* */`.

Never Use `using namespace` in a Header File

This causes the namespace you are `using` to be pulled into the namespace of all files that include the header file. It pollutes the namespace and it may lead to name collisions in the future. Writing `using namespace` in an implementation file is fine though.

Include Guards

Header files must contain a distinctly-named include guard to avoid problems with including the same header multiple times and to prevent conflicts with headers from other projects.

```
#ifndef MYPROJECT_MYCLASS_HPP
#define MYPROJECT_MYCLASS_HPP

namespace MyProject {
  class MyClass {
  };
}

#endif
```

You may also consider using the `#pragma once` directive instead which is quasi-standard across many compilers. It's short and makes the intent clear.

{} Are Required for Blocks.

Leaving them off can lead to semantic errors in the code.

```
// Bad Idea
// This compiles and does what you want, but can lead to confusing
// errors if modification are made in the future and close attention
// is not paid.
for (int i = 0; i < 15; ++i)
  std::cout << i << std::endl;

// Bad Idea
// The cout is not part of the loop in this case even though it appears to be.
int sum = 0;
for (int i = 0; i < 15; ++i)
  ++sum;
  std::cout << i << std::endl;


// Good Idea
// It's clear which statements are part of the loop (or if block, or whatever).
int sum = 0;
for (int i = 0; i < 15; ++i) {
  ++sum;
  std::cout << i << std::endl;
}
```

Keep Lines a Reasonable Length

```
// Bad Idea
// hard to follow
if (x && y && myFunctionThatReturnsBool() && caseNumber3 && (15 > 12 || 2 < 3)) {
}

// Good Idea
// Logical grouping, easier to read
if (x && y && myFunctionThatReturnsBool()
    && caseNumber3
    && (15 > 12 || 2 < 3)) {
}
```

Many projects and coding standards have a soft guideline that one should try to use less than about 80 or 100 characters per line. Such code is generally easier to read. It also makes it possible to have two separate files next to each other on one screen without having a tiny font.

Use "" for Including Local Files

... `<>` is reserved for system includes.

```
// Bad Idea. Requires extra -I directives to the compiler
// and goes against standards.
#include <string>
#include <includes/MyHeader.hpp>

// Worse Idea
// Requires potentially even more specific -I directives and
// makes code more difficult to package and distribute.
#include <string>
#include <MyHeader.hpp>


// Good Idea
// Requires no extra params and notifies the user that the file
// is a local file.
#include <string>
#include "MyHeader.hpp"
```

Initialize Member Variables

...with the member initializer list.

For POD types, the performance of an initializer list is the same as manual initialization, but for other types there is a clear performance gain, see below.

```
// Bad Idea
class MyClass
{
public:
  MyClass(int t_value)
  {
    m_value = t_value;
  }

private:
  int m_value;
};

// Bad Idea
// This leads to an additional constructor call for m_myOtherClass
// before the assignment.
class MyClass
{
public:
  MyClass(MyOtherClass t_myOtherClass)
  {
    m_myOtherClass = t_myOtherClass;
```

```
  }

private:
  MyOtherClass m_myOtherClass;
};

// Good Idea
// There is no performance gain here but the code is cleaner.
class MyClass
{
public:
  MyClass(int t_value)
    : m_value(t_value)
  {
  }

private:
  int m_value;
};

// Good Idea
// The default constructor for m_myOtherClass is never called here, so
// there is a performance gain if MyOtherClass is not is_trivially_default_constructib
le.
class MyClass
{
public:
  MyClass(MyOtherClass t_myOtherClass)
    : m_myOtherClass(t_myOtherClass)
  {
  }

private:
  MyOtherClass m_myOtherClass;
};
```

In C++11 you can assign default values to each member (using `=` or using `{}`).

Assigning default values with =

```
// ... //
private:
  int m_value = 0; // allowed
  unsigned m_value_2 = -1; // narrowing from signed to unsigned allowed
// ... //
```

This ensures that no constructor ever "forgets" to initialize a member object.

Assigning default values with brace initialization

Using brace initialization does not allow narrowing at compile-time.

```
// Best Idea

// ... //
private:
  int m_value{ 0 }; // allowed
  unsigned m_value_2 { -1 }; // narrowing from signed to unsigned not allowed, leads t
o a compile time error
// ... //
```

Prefer `{}` initialization over `=` unless you have a strong reason not to.

Forgetting to initialize a member is a source of undefined behavior bugs which are often extremely hard to find.

If the member variable is not expected to change after the initialization, then mark it `const`.

```
class MyClass
{
public:
  MyClass(int t_value)
    : m_value{t_value}
  {
  }

private:
  const int m_value{0};
};
```

Since a const member variable cannot be assigned a new value, such a class may not have a meaningful copy assignment operator.

Always Use Namespaces

There is almost never a reason to declare an identifier in the global namespace. Instead, functions and classes should exist in an appropriately named namespace or in a class inside of a namespace. Identifiers which are placed in the global namespace risk conflicting with identifiers from other libraries (mostly C, which doesn't have namespaces).

Use the Correct Integer Type for Standard Library Features

The standard library generally uses `std::size_t` for anything related to size. The size of `size_t` is implementation defined.

In general, using `auto` will avoid most of these issues, but not all.

Make sure you stick with the correct integer types and remain consistent with the C++ standard library. It might not warn on the platform you are currently using, but it probably will when you change platforms.

Note that you can cause integer underflow when performing some operations on unsigned values. For example:

```
std::vector<int> v1{2,3,4,5,6,7,8,9};
std::vector<int> v2{9,8,7,6,5,4,3,2,1};
const auto s1 = v1.size();
const auto s2 = v2.size();
const auto diff = s1 - s2; // diff underflows to a very large number
```

Use .hpp and .cpp for Your File Extensions

Ultimately this is a matter of preference, but .hpp and .cpp are widely recognized by various editors and tools. So the choice is pragmatic. Specifically, Visual Studio only automatically recognizes .cpp and .cxx for C++ files, and Vim doesn't necessarily recognize .cc as a C++ file.

One particularly large project (OpenStudio) uses .hpp and .cpp for user-generated files and .hxx and .cxx for tool-generated files. Both are well recognized and having the distinction is helpful.

Never Mix Tabs and Spaces

Some editors like to indent with a mixture of tabs and spaces by default. This makes the code unreadable to anyone not using the exact same tab indentation settings. Configure your editor so this does not happen.

Never Put Code with Side Effects Inside an assert()

```
assert(registerSomeThing()); // make sure that registerSomeThing() returns true
```

The above code succeeds when making a debug build, but gets removed by the compiler when making a release build, giving you different behavior between debug and release builds. This is because `assert()` is a macro which expands to nothing in release mode.

Don't Be Afraid of Templates

They can help you stick to DRY principles. They should be preferred to macros, because macros do not honor namespaces, etc.

Use Operator Overloads Judiciously

Operator overloading was invented to enable expressive syntax. Expressive in the sense that adding two big integers looks like `a + b` and not `a.add(b)`. Another common example is `std::string`, where it is very common to concatenate two strings with `string1 + string2`.

However, you can easily create unreadable expressions using too much or wrong operator overloading. When overloading operators, there are three basic rules to follow as described on stackoverflow.

Specifically, you should keep these things in mind:

- Overloading `operator=()` when handling resources is a must. See Consider the Rule of Zero below.
- For all other operators, only overload them when they are used in a context that is commonly connected to these operators. Typical scenarios are concatenating things with +, negating expressions that can be considered "true" or "false", etc.
- Always be aware of the operator precedence and try to circumvent unintuitive constructs.
- Do not overload exotic operators such as ~ or % unless implementing a numeric type or following a well recognized syntax in specific domain.
- Never overload `operator,()` (the comma operator).
- Use non-member functions `operator>>()` and `operator<<()` when dealing with streams. For example, you can overload `operator<<(std::ostream &, MyClass const &)`

  to enable "writing" your class into a stream, such as `std::cout` or an `std::fstream` or `std::stringstream`. The latter is often used to create a string representation of a value.
- There are more common operators to overload described here.

More tips regarding the implementation details of your custom operators can be found here.

Avoid Implicit Conversions

Single Parameter Constructors

Single parameter constructors can be applied at compile time to automatically convert between types. This is handy for things like `std::string(const char *)` but should be avoided in general because they can add to accidental runtime overhead.

Instead mark single parameter constructors as `explicit`, which requires them to be explicitly called.

Conversion Operators

Similarly to single parameter constructors, conversion operators can be called by the compiler and introduce unexpected overhead. They should also be marked as `explicit`.

```
//bad idea
struct S {
  operator int() {
    return 2;
  }
};
```

```
//good idea
struct S {
  explicit operator int() {
    return 2;
  }
};
```

Consider the Rule of Zero

The Rule of Zero states that you do not provide any of the functions that the compiler can provide (copy constructor, copy assignment operator, move constructor, move assignment operator, destructor) unless the class you are constructing does some novel form of

ownership.

The goal is to let the compiler provide optimal versions that are automatically maintained when more member variables are added.

The original article provides the background, while a follow up article explains techniques for implementing nearly 100% of the time.

Considering Safety

Const as Much as Possible

`const` tells the compiler that a variable or method is immutable. This helps the compiler optimize the code and helps the developer know if a function has a side effect. Also, using `const &` prevents the compiler from copying data unnecessarily. The comments on `const` from John Carmack are also a good read.

```
// Bad Idea
class MyClass
{
public:
  void do_something(int i);
  void do_something(std::string str);
};


// Good Idea
class MyClass
{
public:
  void do_something(const int i);
  void do_something(const std::string &str);
};
```

Carefully Consider Your Return Types

- Getters
  - Returning by `&` or `const &` can have significant performance savings when the normal use of the returned value is for observation
  - Returning by value is better for thread safety and if the normal use of the returned value is to make a copy anyhow, there's no performance lost
  - If your API uses covariant return types, you must return by `&` or `*`
- Temporaries and local values
  - Always return by value.

references: https://github.com/lefticus/cppbestpractices/issues/21 https://twitter.com/lefticus/status/635943577328095232

Do not pass and return simple types by const ref

```
// Very Bad Idea
class MyClass
{
public:
  explicit MyClass(const int& t_int_value)
    : m_int_value(t_int_value)
  {
  }

  const int& get_int_value() const
  {
    return m_int_value;
  }

private:
  int m_int_value;
}
```

Instead, pass and return simple types by value. If you plan not to change passed value, declare them as `const`, but not `const` refs:

```
// Good Idea
class MyClass
{
public:
  explicit MyClass(const int t_int_value)
    : m_int_value(t_int_value)
  {
  }

  int get_int_value() const
  {
    return m_int_value;
  }

private:
  int m_int_value;
}
```

Why? Because passing and returning by reference leads to pointer operations instead by much more faster passing values in processor registers.

Avoid Raw Memory Access

Raw memory access, allocation and deallocation, are difficult to get correct in C++ without risking memory errors and leaks. C++11 provides tools to avoid these problems.

```
// Bad Idea
MyClass *myobj = new MyClass;

// ...
delete myobj;


// Good Idea
auto myobj = std::make_unique<MyClass>(constructor_param1, constructor_param2); // C++
14
auto myobj = std::unique_ptr<MyClass>(new MyClass(constructor_param1, constructor_para
m2)); // C++11
auto mybuffer = std::make_unique<char[]>(length); // C++14
auto mybuffer = std::unique_ptr<char[]>(new char[length]); // C++11

// or for reference counted objects
auto myobj = std::make_shared<MyClass>();

// ...
// myobj is automatically freed for you whenever it is no longer used.
```

Use `std::array` or `std::vector` Instead of C-style Arrays

Both of these guarantee contiguous memory layout of objects and can (and should) completely replace your usage of C-style arrays for many of the reasons listed for not using bare pointers.

Also, avoid using `std::shared_ptr` to hold an array.

Use Exceptions

Exceptions cannot be ignored. Return values, such as using `boost::optional`, can be ignored and if not checked can cause crashes or memory errors. An exception, on the other hand, can be caught and handled. Potentially all the way up the highest level of the application with a log and automatic restart of the application.

Stroustrup, the original designer of C++, makes this point much better than I ever could.

Use C++-style cast instead of C-style cast

Use the C++-style cast (static_cast<>, dynamic_cast<> ...) instead of the C-style cast. The C++-style cast allows more compiler checks and is considerably safer.

```
// Bad Idea
double x = getX();
int i = (int) x;

// Not a Bad Idea
int i = static_cast<int>(x);
```

Additionally the C++ cast style is more visible and has the possibility to search for.

But consider refactoring of program logic (for example, additional checking on overflow and underflow) if you need to cast `double` to `int`. Measure three times and cut 0.9999999999981 times.

Do not define a variadic function

Variadic functions can accept a variable number of parameters. The probably best known example is printf(). You have the possibility to define this kind of functions by yourself but this is a possible security risk. The usage of variadic functions is not type safe and the wrong input parameters can cause a program termination with an undefined behavior. This undefined behavior can be exploited to a security problem. If you have the possibility to use a compiler that supports C++11, you can use variadic templates instead.

It is technically possible to make typesafe C-style variadic functions with some compilers

Additional Resources

How to Prevent The Next Heartbleed by David Wheeler is a good analysis of the current state of code safety and how to ensure safe code.

Considering Maintainability

Avoid Compiler Macros

Compiler definitions and macros are replaced by the preprocessor before the compiler is ever run. This can make debugging very difficult because the debugger doesn't know where the source came from.

```
// Bad Idea
#define PI 3.14159;

// Good Idea
namespace my_project {
  class Constants {
  public:
    // if the above macro would be expanded, then the following line would be:
    //   static const double 3.14159 = 3.14159;
    // which leads to a compile-time error. Sometimes such errors are hard to understa
nd.
    static constexpr double PI = 3.14159;
  };
}
```

Consider Avoiding Boolean Parameters

They do not provide any additional meaning while reading the code. You can either create a separate function that has a more meaningful name, or pass an enumeration that makes the meaning more clear.

See http://mortoray.com/2015/06/15/get-rid-of-those-boolean-function-parameters/ for more information.

Avoid Raw Loops

Know and understand the existing C++ standard algorithms and put them to use. See C++ Seasoning for more details.

Never Use `assert` With Side Effects

```
// Bad Idea
assert(set_value(something));

// Better Idea
[[maybe_unused]] const auto success = set_value(something);
assert(success);
```

The `assert()` will be removed in release builds which will prevent the `set_value` call from every happening.

So while the second version is uglier, the first version is simply not correct.

Properly Utilize 'override' and 'final'

These keywords make it clear to other developers how virtual functions are being utilized, can catch potential errors if the signature of a virtual function changes, and can possibly hint to the compiler of optimizations that can be performed.

Considering Portability

Know Your Types

Most portability issues that generate warnings are because we are not careful about our types. Standard library and arrays are indexed with `size_t`. Standard container sizes are reported in `size_t`. If you get the handling of `size_t` wrong, you can create subtle lurking 64-bit issues that arise only after you start to overflow the indexing of 32-bit integers. char vs unsigned char.

http://www.viva64.com/en/a/0010/

Other Concerns

Most of the other concerns in this document ultimately come back to portability issues. Avoid statics is particularly of note.

Considering Threadability

Avoid Global Data

Global data leads to unintended side effects between functions and can make code difficult or impossible to parallelize. Even if the code is not intended today for parallelization, there is no reason to make it impossible for the future.

Statics

Besides being global data, statics are not always constructed and deconstructed as you would expect. This is particularly true in cross-platform environments. See for example, this g++ bug regarding the order of destruction of shared static data loaded from dynamic modules.

Shared Pointers

`std::shared_ptr` is "as good as a global" (http://stackoverflow.com/a/18803611/29975) because it allows multiple pieces of code to interact with the same data.

Singletons

A singleton is often implemented with a static and/or `shared_ptr`.

Avoid Heap Operations

Much slower in threaded environments. In many or maybe even most cases, copying data is faster. Plus with move operations and such and things.

Mutex and mutable go together (M&M rule, C++11)

For member variables it is good practice to use mutex and mutable together. This applies in both ways:

- A mutable member variable is presumed to be a shared variable so it should be

  synchronized with a mutex (or made atomic)
- If a member variable is itself a mutex, it should be mutable. This is required to use it inside a const member function.

For more information see the following article from Herb Sutter:
http://herbsutter.com/2013/05/24/gotw-6a-const-correctness-part-1-3/

See also related safety discussion about `const &` return values

Considering Performance

Build Time

Forward Declare When Possible

This:

```
// some header file
class MyClass;

void doSomething(const MyClass &);
```

instead of:

```
// some header file
#include "MyClass.hpp"

void doSomething(const MyClass &);
```

This applies to templates as well:

```
template<typename T> class MyTemplatedType;
```

This is a proactive approach to reduce compilation time and rebuilding dependencies.

Avoid Unnecessary Template Instantiations

Templates are not free to instantiate. Instantiating many templates, or templates with more code than necessary increases compiled code size and build time.

For more examples see this article.

Avoid Recursive Template Instantiations

Recursive template instantiations can result in a significant load on the compiler and more difficult to understand code.

Consider using variadic expansions and folds when possible instead.

Analyze the Build

The tool Templight can be used to analyze the build time of your project. It takes some effort to get built, but once you do, it's a drop in replacement for clang++.

After you build using Templight, you will need to analyze the results. The templight-tools project provides various methods. (Author's Note: I suggest using the callgrind converter and visualizing the results with kcachegrind).

Firewall Frequently Changing Header Files

Don't Unnecessarily Include Headers

The compiler has to do something with each include directive it sees. Even if it stops as soon as it seems the `#ifndef` include guard, it still had to open the file and begin processing it.

include-what-you-use is a tool that can help you identify which headers you need.

Reduce the load on the preprocessor

This is a general form of "Firewall Frequently Changing Header Files" and "Don't Unnecessarily Include Headers." Tools like BOOST_PP can be very helpful, but they also put a huge burden on the preprocessor.

Consider using precompiled headers

The usage of precompiled headers can considerably reduce the compile time in large projects. Selected headers are compiled to an intermediate form (PCH files) that can be faster processed by the compiler. It is recommended to define only frequently used header that changes rarely as precompiled header (e.g. system and library headers) to achieve the compile time reduction. But you have to keep in mind, that using precompiled headers has several disadvantages:

- The usage of precompiled header is not portable.
- The generated PCH files are machine dependent.
- The generated PCH files can be quite large.
- It can break your header dependencies. Because of the precompiled headers, every file has the possibility to include every header that is marked as a precompiled header. In result it can happen, that the build fails if you disable the precompiled headers. This can be an issue if you ship something like a library. Because of this it is highly recommend to build once with precompiled header enabled and a second time without them.

Precompiled headers is supported by the most common compiler, like GCC, Clang and Visual Studio. Tools like cotire (a plugin for cmake) can help you to add precompiled headers to your build system.

Consider Using Tools

These are not meant to supersede good design

- ccache
- warp, Facebook's preprocessor

Put tmp on Ramdisk

See this YouTube video for more details.

Use the gold linker

If on Linux, consider using the gold linker for GCC.

Runtime

Analyze the Code!

There's no real way to know where your bottlenecks are without analyzing the code.

- http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/
- http://www.codersnotes.com/sleepy

Simplify the Code

The cleaner, simpler, and easier to read the code is, the better chance the compiler has at implementing it well.

Use Initializer Lists

```
// This
std::vector<ModelObject> mos{mo1, mo2};

// -or-
auto mos = std::vector<ModelObject>{mo1, mo2};
```

```
// Don't do this
std::vector<ModelObject> mos;
mos.push_back(mo1);
mos.push_back(mo2);
```

Initializer lists are significantly more efficient; reducing object copies and resizing of containers.

Reduce Temporary Objects

```
// Instead of
auto mo1 = getSomeModelObject();
auto mo2 = getAnotherModelObject();

doSomething(mo1, mo2);
```

```
// consider:

doSomething(getSomeModelObject(), getAnotherModelObject());
```

This sort of code prevents the compiler from performing a move operation...

Enable move operations

Move operations are one of the most touted features of C++11. They allow the compiler to avoid extra copies by moving temporary objects instead of copying them in certain cases.

Certain coding choices we make (such as declaring our own destructor or assignment operator or copy constructor) prevents the compiler from generating a move constructor.

For most code, a simple

```
ModelObject(ModelObject &&) = default;
```

would suffice. However, MSVC2013 doesn't seem to like this code yet.

Kill `shared_ptr` Copies

`shared_ptr` objects are much more expensive to copy than you'd think they would be. This is because the reference count must be atomic and thread-safe. So this comment just re-enforces the note above: avoid temporaries and too many copies of objects. Just because we are using a pImpl it does not mean our copies are free.

Reduce Copies and Reassignments as Much as Possible

For more simple cases, the ternary operator can be used:

```
// Bad Idea
std::string somevalue;

if (caseA) {
  somevalue = "Value A";
} else {
  somevalue = "Value B";
}
```

```
// Better Idea
const std::string somevalue = caseA ? "Value A" : "Value B";
```

More complex cases can be facilitated with an immediately-invoked lambda.

```
// Bad Idea
std::string somevalue;

if (caseA) {
  somevalue = "Value A";
} else if(caseB) {
  somevalue = "Value B";
} else {
  somevalue = "Value C";
}
```

```
// Better Idea
const std::string somevalue = [&](){
    if (caseA) {
      return "Value A";
    } else if (caseB) {
      return "Value B";
    } else {
      return "Value C";
    }
  }();
```

Avoid Excess Exceptions

Exceptions which are thrown and captured internally during normal processing slow down the application execution. They also destroy the user experience from within a debugger, as debuggers monitor and report on each exception event. It is best to just avoid internal

exception processing when possible.

Get rid of “new”

We already know that we should not be using raw memory access, so we are using `unique_ptr` and `shared_ptr` instead, right? Heap allocations are much more expensive than stack allocations, but sometimes we have to use them. To make matters worse, creating a `shared_ptr` actually requires 2 heap allocations.

However, the `make_shared` function reduces this down to just one.

```
std::shared_ptr<ModelObject_Impl>(new ModelObject_Impl());

// should become
std::make_shared<ModelObject_Impl>(); // (it's also more readable and concise)
```

Prefer `unique_ptr` to `shared_ptr`

If possible use `unique_ptr` instead of `shared_ptr`. The `unique_ptr` does not need to keep track of its copies because it is not copyable. Because of this it is more efficient than the `shared_ptr`. Equivalent to `shared_ptr` and `make_shared` you should use `make_unique` (C++14 or greater) to create the `unique_ptr`:

```
std::make_unique<ModelObject_Impl>();
```

Current best practices suggest returning a `unique_ptr` from factory functions as well, then converting the `unique_ptr` to a `shared_ptr` if necessary.

```
std::unique_ptr<ModelObject_Impl> factory();

auto shared = std::shared_ptr<ModelObject_Impl>(factory());
```

Get rid of std::endl

`std::endl` implies a flush operation. It's equivalent to `"\n" << std::flush`.

Limit Variable Scope

Variables should be declared as late as possible, and ideally only when it's possible to initialize the object. Reduced variable scope results in less memory being used, more efficient code in general, and helps the compiler optimize the code further.

```
// Good Idea
for (int i = 0; i < 15; ++i)
{
  MyObject obj(i);
  // do something with obj
}

// Bad Idea
MyObject obj; // meaningless object initialization
for (int i = 0; i < 15; ++i)
{
  obj = MyObject(i); // unnecessary assignment operation
  // do something with obj
}
// obj is still taking up memory for no reason
```

This topic has an associated discussion thread.

Prefer `double` to `float`, But Test First

Depending on the situation and the compiler's ability to optimize, one may be faster over the other. Choosing `float` will result in lower precision and may be slower due to conversions. On vectorizable operations `float` may be faster if you are able to sacrifice precision.

`double` is the recomended default choice as it is the default type for floating point values in C++.

See this stackoverflow discussion for some more information.

Prefer `++i` to `i++`

... when it is semantically correct. Pre-increment is faster than post-increment because it does not require a copy of the object to be made.

```
// Bad Idea
for (int i = 0; i < 15; i++)
{
  std::cout << i << '\n';
}

// Good Idea
for (int i = 0; i < 15; ++i)
{
  std::cout << i << '\n';
}
```

Even if many modern compilers will optimize these two loops to the same assembly code, it is still good practice to prefer `++i`. There is absolutely no reason not to and you can never be certain that your code will not pass a compiler that does not optimize this. You should be also aware that the compiler will not be able optimize this only for integer types and not necessarily for all iterator or other user defined types.
The bottom line is that it is always easier and recommended to use the pre-increment operator if it is semantically identical to the post-increment operator.

Char is a char, string is a string

```
// Bad Idea
std::cout << someThing() << "\n";

// Good Idea
std::cout << someThing() << '\n';
```

This is very minor, but a `"\n"` has to be parsed by the compiler as a `const char *` which has to do a range check for `\0` when writing it to the stream (or appending to a string). A '\n' is known to be a single character and avoids many CPU instructions.

If used inefficiently very many times it might have an impact on your performance, but more importantly thinking about these two usage cases gets you thinking more about what the compiler and runtime has to do to execute your code.

Never Use `std::bind`

`std::bind` is almost always way more overhead (both compile time and runtime) than you need. Instead simply use a lambda.

```
// Bad Idea
auto f = std::bind(&my_function, "hello", std::placeholders::_1);
f("world");

// Good Idea
auto f = [](const std::string &s) { return my_function("hello", s); };
f("world");
```

Enable Scripting

The combination of scripting and compiled languages is very powerful. It gives us the things we've come to love about compiled languages: type safety, performance, thread safety options, consistent memory model while also giving us the flexibility to try something new quickly without a full rebuild.

The VM based compiled languages have learned this already: JRuby, Jython, IronRuby, IronPython

- ChaiScript
- AngelScript
- luabind
- sol2 (bindings for Lua)
- SWIG (simplified wrapper and interface generator)
- pybind11 (Python and modern C++ interoperability)

Further Reading

Note: This book has now inspired a video series from O'Reilly, Learning C++ Best Practices

- https://github.com/isocpp/CppCoreGuidelines The C++ Core Guidelines are a set of tried-and-true guidelines, rules, and best practices about coding in C++
- https://www.gitbook.com/book/alexastva/the-ultimate-question-of-programming-refactoring-/details - The Ultimate Question of Programming, Refactoring, and Everything
- http://llvm.org/docs/CodingStandards.html - LLVM Coding Standards - very well written
- http://geosoft.no/development/cppstyle.html
- https://google.github.io/styleguide/cppguide.html (Note that Google's standard document makes several recommendations which we will NOT be following. For example, they explicitly forbid the use of exceptions, which makes RAII impossible.)
- https://isocpp.org/faq/
- http://www.cplusplus.com/
- http://www.gamasutra.com/view/news/128836/InDepth_Static_Code_Analysis.php - Article from John Carmack on the advantages of static analysis
- https://svn.boost.org/trac/boost/wiki/BestPracticeHandbook - Best Practice Handbook from Nial Douglas
- http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=ListOfChecks
- http://emptycrate.com/
- http://stackoverflow.com/questions/tagged/c%2b%2b-faq?sort=votes&pageSize=15 - StackOverflow C++ FAQ
- http://codergears.com/qacenter/ discussion center for C and C++ best practices
- http://www.viva64.com/en/b/0391/ The Ultimate Question of Programming, Refactoring, and Everything

Final Thoughts

Expand your horizons and use other programming languages. Other languages have different constructs and expressions. Learning what else is out there will encourage you to be more creative with your C++ and write cleaner, more expressive code.